Following are some hints and tips that might be useful in solving this assignment.

Standard Deep Learning Models

- This dataset is so easy that any standard deep learning based object detection or semantic segmentation model should be able to achieve the required accuracies using pretty much its standard configuration without requiring a lot of tweaking from you.
    - for object detection, I'd recommend the *torch.hub* version of *YOLOv5* which is very easy to use and is known to work very well on this assignment since more than 75% of the students in last year used it
    - for semantic segmentation, the simplest option would be to just use the UNet in the lecture notebooks
    - for instance segmentation, I'd recommend *YOLOv8*
- If you have a working object detector, you can simplify the semantic segmentation task
    - train a 2-class (foreground-background) segmentation model on the 28 x 28 masks corresponding to individual digits instead of a 11-class model on the combined 64 x 64 mask for the entire image
    - once you have the two bounding boxes from the detector, you can use the corresponding image patches as input to your segmentation model to obtain the mask for each digit and it should be trivial to combine the two masks to create the combined 64 x 64 mask
- If you have a working semantic segmentation model, you don't necessarily need a separate object detector since boxes can be inferred from segmentation masks
    - denoise the images
    - use *np.unique* or histogram analysis to find all digits in the mask (ignoring background pixels)
        - it will also give you the number of times each digit occurs in the mask
    - find the two digits with the maximum number of mask pixels
    - for each of these two digits:
        - compute the average of the x and y coordinates of the pixel locations where it occurs which will give you the centroid of the bounding box
        - construct of 28 x 28 box around the centroid

- If your chosen model is not working, the strong likelihood is that there is a bug in your data processing rather than anything to do with the model itself.
    - It would therefore be a good idea to stick with your model instead of switching to a different one and starting all over again.
    - Also I would not recommend making big changes to standard architectures since most such changes are likely to degrade its performance rather than improve it
    - If you must tweak the model, stick to minor stuff like changing the learning rate or batch size or trying out augmentation techniques
- If your model is training too slowly or taking so much GPU RAM that you are running out of memory on Colab, it is likely to be using one of the larger backbones like ResNet50 / 101 which are a huge overkill for this dataset so I'd recommend using the smallest backbone that your model supports
- Memory consumption is also influenced by the size of the input images and most models are designed to handle much larger images around 500 - 1000 pixels in each dimension which is again an overkill for the tiny 64 x 64 images in this dataset so the easiest way to reduce memory consumption would be to adapt the input layer for 64 x 64 images
    - in case of an object detector, it might also help to adapt the output layer to produce exactly two boxes instead of arbitrary number of boxes that most detectors are designed for

Simplifying detection to classification

If you find that training the detector is too hard, you can even solve the detection problem on this dataset as a classification problem because of all the constraints that you are provided with.
Following are two different approaches that I can suggest:

Sliding window classification

- extract all possible 28 x 28 patches from each 64 x 64 image and pass each one of these through an MNIST classifier (like the ones from assignments 1 and 2)
    - using strides of 1 and 2 would respectively give you 36 x 36 = 1,296 and 18 x 18 = 324 sliding windows
    - If you stack these patches together and pass them through your model in a single forward pass, I'm sure it will be able to work at >> 10 FPS (at least with stride 2 which is enough to get >85% IOU)
    - you can use a library like patchify to generate the patches
- my guess would be the output probability distribution will be sufficient to distinguish between background and foreground patches
    - patches with a single digit are likely to have a strongly unimodal distribution (i.e. one probability is much higher than all others)
    - patches with overlapping digits might have a bimodal distribution
    - background patches are likely to have something closer to a uniform distribution
- in the worst case, you may have to retrain the classifier to do 11-class classification with background patches added to your training set as a separate class

Direct classification of the bounding box corner

- since each bounding box is 28 x 28 and all bounding boxes are contained entirely inside the 64 x 64 images, there are only 64 -28 = 36 possible values that the x and y coordinates of the top left (or any other) bounding box corner can take
- predicting these coordinates can therefore be treated as a couple of 36-class classification problems
- it should be easy to modify the classification CNN from assignment 2 to output four 36-class probability distributions (2 for each digit) in addition to a couple of 10-class distributions for the digits themselves
- this might seem like a strange idea but it works very well in practice and was in fact the best performing detection model last year

Image processing

There are so many constraints in this dataset that you strictly don't even need to use deep learning to solve most of this assignment.
Following is a simple idea off the top of my head (and I'm sure there are many others) that only needs 206-level image processing techniques for detection and segmentation (along with a simple classifier)

- there are exactly 3 colors in each image so if you plot the histogram (RGB or greyscale), it will be strongly tri-modal
- you can cluster the pixel values (e.g. using K-Means) into three clusters which will very likely give you these colors in most cases (possibly not in the most tricky cases with high overlap)
    - the largest cluster will correspond to the background since most pixels are background and the other two clusters will correspond to the two digits
- once you have the colors, you can use *np.argwhere* to find the pixel values that have these colors which will give you the approximate masks for the digits
- averaging the x and y coordinates of these pixels will give you the centroid of each digit and you can then construct a 28 x 28 box around each
- for classification, you can use the networks from assignment 1 or 2